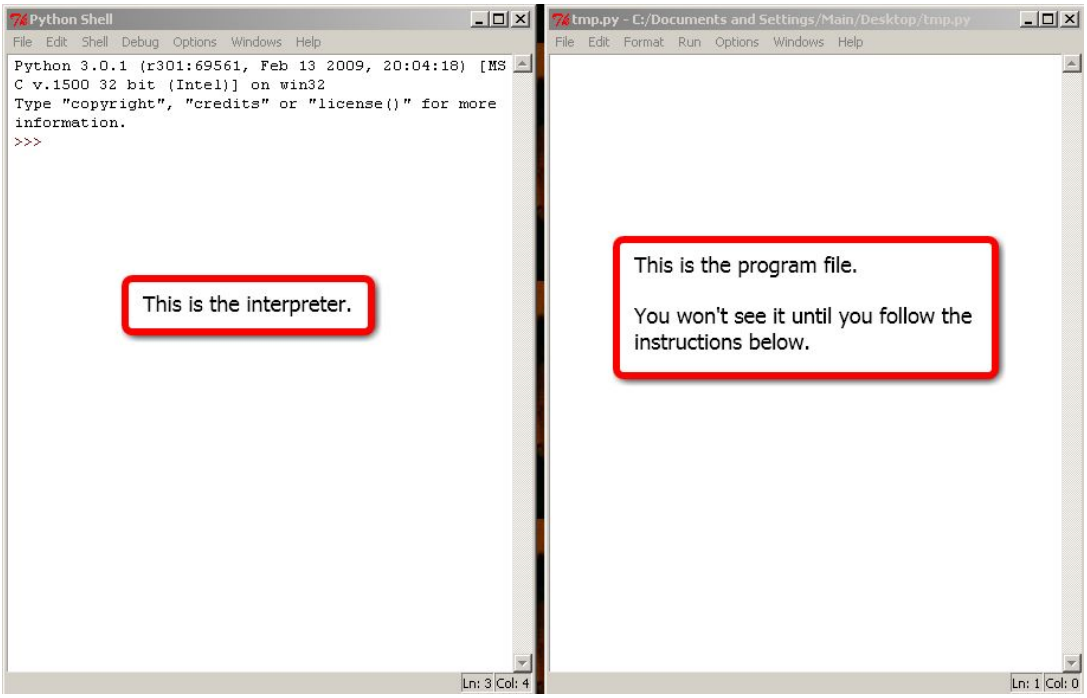
Unit 1

Learn something new

1. Start the IDLE environment for writing Python programs. What you get should look like the left-hand window in the following picture.



This window says "Python Shell" at the top to tell us that it is the Python **interpreter**, that is, a place where we can type Python instructions and have them carried out. The ">>>" is a **prompt** that marks the place to type.

Most often, though, we'll want to carry out more than just one instruction at a time. Rather than giving instructions directly to the interpreter, we'll save them in a file and pass a whole collection of instructions to the interpreter all at once. To open a new **program file**, choose "file | new window" in the interpreter window; even better, use the shortcut Ctrl+N. The program file should look like the right-hand window in the picture.

Before you do anything else, give your program file a name by choosing "file | save as." Be sure to choose a name ending in '.py'

2. You're now ready to start using Python. At the ">>>" prompt in the interpreter window, type "print(5)" and hit enter. The result should look like this.

```
>>> print(5)
```

```
5
>>>
```

You told the interpreter to print the number 5 and it did.

3. Giving an instruction directly to the interpreter can be useful for a quick experiment. More typically, though, we'll save instructions in a file. Type the following in the program file window.

```
print(5)
print(6)
```

This is a **program**, a set of instructions for a computer written in a programming language. Before you can **run** a program—asking the interpreter to carry out the instructions you've written—you first need to save it. Choose "file | save" in the program file window or, even better, use the shortcut Ctrl+S. Then run the program by choosing "run | run module" or, preferably, using the shortcut key F5, which is located at the top of the keyboard. The following should then appear in the *interpreter* window.

```
>>>
5
6
>>>
```

Congratulations. You've written and run your first Python program.

From now on, when we ask you to run a program: (a) type the program in the program file (b) save it with Ctrl+S and (c) run it with F5. You must be using the program file window—not the interpreter window—when you type Ctrl+S and F5.

4. The program we've just written is pretty boring. As a first step toward something more interesting, edit the program file now so that it contains just the following two lines. Note that you'll have to delete the old lines in the file; in general, your programs should match our examples *exactly*—don't include anything other than what's shown, use the same punctuation, same layout (spacing and indentation), etc.

```
student = 'Fred'
print('Hello', student)
```

The first line of this program creates a **string**—a sequence of characters, in this case F-r-e-d—and gives it a name: `student`. To indicate a string, we enclose it in quotes.

Be careful not to be thrown off by the equals sign. It doesn't mean the same thing as in an algebra class or even in other programming languages like C++ or Java.

The equals sign is used to say that a name refers (at least temporarily) to an **object**, some piece of data that Python can process. In this case, `student` is just a name for the string object `'Fred'`. It's a good idea to get in the habit of reading this without using the word 'equals'. Say, for example, "student is the name for the string Fred," or "student is Fred."

In the second line above, we tell Python to print the string `'Hello'` followed by the string that `student` refers to. Run the program (Crtl+S, then F5) and check that the following appears in the interpreter window.

```
>>>
Hello Fred
>>>
```

5. Edit the program file by adding two lines, as follows.

```
student = 'Fred'
print('Hello', student)
student = 'Ted'
print('Hello', student)
```

Here, after greeting Fred, we reassign the name `student` to refer to a new string and use it to say hello to Ted. Run the program and check that you get the following result.

```
Hello Fred
Hello Ted
```

To save space we've stopped showing the ">>>" prompts.

6. Now suppose we have a long list of students to greet. Rather than writing two nearly identical lines many times, we'd like to tell Python to go through the list doing just what we've done. That is, the interpreter should assign `student` one by one to a series

of names and for each assignment it should carry out the statement `print('Hello', student)`. Here's the program we want; edit the program file to match it.

```
for student in ['Fred', 'Ted', 'Ed']:
   print('Hello', student)
```

Run the program and check that the output looks like this.

```
Hello Fred
Hello Ted
Hello Ed
```

Clearly this is a much nicer way to print a number of greetings. It's also clear how to add more students; we could just change `['Fred', 'Ted', 'Ed']` to, say, `['Fred', 'Ted', 'Ed', 'Jennifer']`.

Here are a few important notes about the `for` statement we've just written.

6.1. To specify the series of objects `student` should refer to, we provided a **list**, in this case a list consisting of three string objects. A list is a sequence of objects. We write one in Python by separating the objects with commas and enclosing the sequence with square brackets. Here's a list of two strings: `['cat', 'dog']`

6.2. A list is itself a kind of object in Python, along with **integers** like 5—numbers without decimals—and strings like `'Fred'`. So we can have a list of three objects that looks like this: `[5, 'Fred', [7, 19]]`. The first item in this list is the integer 5, the second is the string `'Fred'` and the third is a list of two more integers.

6.3. The first line of our program ends with a colon (:). The colon is necessary; in a `for` statement it marks the end of the part that specifies *when* to carry out the instructions that follow. In our case, we want them carried out once for each name in the list.

6.4. The second part of the `for` statement must be indented. Indented statements after the first line of a `for` statement specify the instructions to be carried out for each assignment. In our case, we included just one instruction—`print('Hello', student)`—but we can add as many more as we like so long as they are all indented equally. IDLE helps by indenting automatically when we hit enter after the colon.

7. We can make our program a little clearer, without changing what it does, by using a name for the list of students. Edit as follows and run the program to make sure the output is unchanged.

```
students = ['Fred', 'Ted', 'Ed']
for student in students:
   print('Hello', student)
```

In this case, we use the name **students** to refer to the list ['Fred', 'Ted', 'Ed']. We use the other name, **student**, to refer one at a time to each of the strings inside of **students**.

Note that the Python interpreter has no idea that there is any connection between the names **student** and **students**. We chose these two related names to help make our program clearer. Using sensible names is an important technique in programming. The interpreter is just as happy if we write our program as follows. It runs exactly the same as the previous version…but it makes life for *us* much less pleasant.

```
xy34q = ['Fred', 'Ted', 'Ed']
for prz in xy34q:
   print('Hello', prz)
```

8. Here's a very similar program. So far, this uses nothing new:

```
meat = 'ham'
breads = ['rye', 'whole wheat', 'a roll']
for bread in breads:
   print(meat, 'on', bread)
```

Run it and you'll find that it lists three kinds of ham sandwiches. Make sure you understand how the program works before you continue.

Now suppose we were to repeat the very same instructions, changing the kind of meat to pastrami the second time through. *Don't do this*; just think about it for a moment. Here's what the program would look like.

```
meat = 'ham'
breads = ['rye', 'whole wheat', 'a roll']
for bread in breads:
```

```
    print(meat, 'on', bread)

meat = 'pastrami'
breads = ['rye', 'whole wheat', 'a roll']
for bread in breads:
    print(meat, 'on', bread)
```

This would list three kinds of ham sandwiches and then three kinds of pastrami sandwiches. But this should look familiar. If we want **meat** to take on successive values and do the same work with each, it would be better to use a **for** statement. Here's the result. *Do* edit the program now to match the following and then run it. The output should list 12 kinds of sandwiches, 3 for each kind of meat.

```
meats = ['ham', 'pastrami', 'roast beef', 'chicken']
breads = ['rye', 'whole wheat', 'a roll']
for meat in meats:
    for bread in breads:
        print(meat, 'on', bread)
```

Again, there are a few important points to note.

8.1. This program is easiest to understand from the bottom up.

- The last line prints one line, assuming **meat** and **bread** have been assigned.
- This line is indented under a **for** statement that assigns values one at a time to **bread** from a list of three possibilities. So the effect of the second **for** is to print three lines.
- But this whole **for**—the one involving bread—is indented beneath one that assigns values to **meat** one at a time from a list of four. So the effect is to print three lines four times, for a total of twelve.

8.2. If we had just mechanically modified the code in Step 7, we would have gotten the following. *Don't do this*; just consider it.

```
meats = ['ham', 'pastrami', 'roast beef','chicken']
for meat in meats:
    breads = ['rye', 'whole wheat', 'a roll']
    for bread in breads:
        print(meat, 'on', bread)
```

This works fine. For each kind of meat, it sets up the list of breads and then prints three lines. The problem is just that it does unnecessary work. Since the

list of breads is always the same, it's silly to assign it four times. The earlier version is more efficient.

9. We've been using **for** to do work repeatedly, once for each item in a list. But **for** works just as well with other ordered collections. For example, a string is an ordered collection of characters. So we can write the following to print each letter in a string separately. Try it.

```
vowels = 'aeiou'
for letter in vowels:
   print(letter)
```

We can also use the word 'in' very naturally in Python to see if an item is contained in a collection.

```
if 'a' in 'pineapple':
   print('found a')
if 'b' in 'pineapple':
   print('found b')
```

This prints 'found a', but not 'found b'. Notice that the form here is very much like what we've seen before. The line that ends in a colon specifies *when* to execute the indented statements that come after it. Before we used **for** to say: "Do this for each value in the list." Now we're using **if** to say: "Do this provided a certain condition holds."

Putting **in** and **if** together with what we've already seen, we get a program that reports which vowels are found in the word 'pineapple.' Try it.

```
vowels = 'aeiou'
word = 'pineapple'
for letter in vowels:
   if letter in word:
      print(letter, 'is in', word)
```